Abstract—Because program behaviors of database applications depend on the data used, code coverages do not effectively test database applications. Additionally, test coverages for database applications that focus on predicates in Structured Query Language (SQL) queries are not useful if the necessary predicates are omitted. In this paper, we present two new database applications using Plain Pairwise Coverage (PPC) and Selected Pairwise Coverage (SPC) for SQL queries called Plain Pairwise Coverage Testing (PPCT) and Selected Pairwise Coverage Testing (SPCT), respectively. These coverages are based on pairwise testing coverage, which employs selected elements in the SQL SELECT query as parameters. We also implement a coverage calculation tool and conduct case studies on two open source software systems. PPCT and SPCT can detect many bugs, which are not detected by existing test methods based on predicates in the query. Furthermore, the case study suggests that SPCT can detect bugs more efficiently than PPCT and the costs of SPCT can be further reduced by ignoring records filtered out by the conditions of the query.

Index Terms—combinatorial testing; database testing; coverage; test data

I. INTRODUCTION

Testing database applications using components based on database management systems is difficult because the execution path and behaviors depend on the database state. Although some code coverages (e.g., statement coverage, branch coverage, etc. [1][2]) and testing methods automatically generate tests, prioritize test cases, and localize bugs based on code coverage [3][4][5][6], they cannot detect bugs due to the records in the databases because data variations in the records are not considered.

Most applications maintain data using a Relational Database Management System (RDBMS) and a Structured Query Language (SQL) query to manage the contained data. Some coverage and test methods consider data variations with SQL executed in the applications [7][8][9], but they focus on the condition predicates in the query such as in the JOIN or WHERE clauses. These methods can detect bugs caused by conditional mistakes in the queries [e.g., misplaced logical operators (AND, OR) or mistakes of conditional inversion]. However, they cannot detect bugs due to the lack of conditions such as an omission of a necessary condition in the SQL query or in the IF statement in the program code.

To detect bugs due to a lack of conditions, we propose Plain Pairwise Coverage Testing (PPCT) and Selected Pairwise Coverage Testing (SPCT), which are new testing methods for database applications using Plain Pairwise Coverage (PPC) and Selected Pairwise Coverage (SPC), respectively, for SQL queries. Pairwise testing [10] is a Combinatorial Test Design technique in which two parameters and their test values are taken as inputs to test all combinations of two parameters’ values. In PPC, all COLUMNs with parameters in the TABLEs in the select query are used as the parameters, and the test values are set to verify if all possible combinations of the two parameters’ values appear in the query results. In SPC, COLUMNs in the SELECT query are used as the parameters, and all their test values are set to verify if all possible combinations of the two parameters’ values appear in the query results. Furthermore, SPCT focuses on the WHERE clauses to remove combinations that cannot be obtained by the query.

This study aims to address the following Research Questions:

RQ1 Can PPCT and SPCT detect bugs that existing testing method cannot?
RQ2 How many records are needed to reach 100% PPC and SPC coverage?

To evaluate the applicability of PPCT and SPCT, we use a case study where PPCT and SPCT are applied to two open source software systems [11][12]. Each open source software system has about 400KLOC and interacts with about 100 tables and 1000 columns. Because they have bug management systems, we employ their issues as target bugs. In our case study, PPCT and SPCT detects 61 of the 89 bugs that existing method cannot. In addition, the case study results suggest that ignoring records not obtained by the query can further reduce costs.

This paper makes the followings contributions:

- PPC and SPC, which are new record coverages that apply pairwise testing coverage using selected elements in a query, are proposed.
- A tool to calculate PPC and SPC is implemented.
- PPCT and SPCT, which are new test methods for database applications using PPC and SPC, respectively, are proposed.
• PPCT and SPCT are applied to two actual products.
• PPCT and SPCT can detect 68.54(%) of bugs that existing testing method cannot.
• SPCT can detect bugs more efficiently than PPCT by ignoring redundant combinations.

The remainder of the paper is organized as follows. Section II provides background information and a motivating example. Section III describes PPCT, SPCT, and our tool to calculate PPC and SPC. Sections IV and V present our case study and results, respectively. Section VI discusses related works. Finally, Section VII contains our conclusion and future works.

II. BACKGROUND

Table I shows the schema, while Figure 1 shows the code and query of our motivating example. Every record in the Table item, which indicates whether the record is alive or not, has the column voided. The method sumPrice(String category) calculates the total price of the ‘unvoided’ items. This code finds items by category (Line 4) and sums up their prices (Line 6-10). Because the code sums only ‘unvoided’ item’s prices, ‘voided’ items in the query must be filtered out. However, ‘voided’ items are not filtered because there is neither WHERE clause in the query nor an IF statement in the code. Thus, records must be prepared according to specific criteria to test this code.

A. Testing with Code Coverage

Consider a test that uses code coverage (e.g., statement coverage). In the motivating example, we only need sumPrice("food") for the item record whose category = 'food' to achieve 100% statement coverage. However, this method cannot detect the abovementioned bug with a record whose category = 'food' AND voided = false even though the statement coverage is 100%.

B. Testing with Predicate Coverage of SQL

Some studies test database applications with SQL [7][8][9] by focusing on the predicates of SQL such as conditions in the WHERE clause and the integrity constraints of the database schema. These criteria can detect conditional mistakes in the query because they verify whether record variations can be covered in terms of the conditions. However, they cannot detect bug in the motivating example because it is due to the lack of a condition.

Consider a code test using the ‘Full Predicate Coverage’(FPC) proposed by Tuya et al. [8]. This coverage focuses on the conditions of the JOIN clause and the WHERE clause in the query to evaluate the MC/DC coverage of the query. For the motivating example, FPC focuses on the condition WHERE category = 'food'. To achieve 100% coverage, FPC requires records whose category = ‘food’ and category ≠ ‘food’. However, even with 100% coverage, it cannot detect bugs with records voided = false.

C. Pairwise Testing

Combinatorial testing can detect bugs due to parameter interactions in the software being tested with a covering array test suite generated by an algorithm. Pairwise testing is a combinatorial testing technique in which two parameters and all their test values are used as inputs to test all possible combinations. If only one or two conditions exist, more than 70% of bugs can be detected [13]. Because pairwise-generated test suites are smaller than exhaustive ones and effectively find defects, many tools exist to generate pairwise test suites [14].

To test database applications with a pairwise testing technique, the parameters are (tables or columns), candidate values, and presence of constraints all must be considered.

III. PAIRWISE COVERAGE-BASED TESTING

PPCT and SPCT are new test methods for database applications that focus on the variety of database records used in testing. To reduce the number of tested combinations, we use a pairwise testing technique.

A. Overview

Figure 2 overviews PPCT and SPCT. The target of PPCT and SPCT is the database used in the application being tested. Three inputs are required: the target database for the coverage measurement, a test case to extract the SQL SELECT query, and the test criteria. The last one is retrieved from the

<table>
<thead>
<tr>
<th>Table I</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Attribute</strong></td>
</tr>
<tr>
<td>id</td>
</tr>
<tr>
<td>category</td>
</tr>
<tr>
<td>voided</td>
</tr>
<tr>
<td>price</td>
</tr>
<tr>
<td>created_at</td>
</tr>
<tr>
<td>updated_at</td>
</tr>
</tbody>
</table>

/* Calculate a total price of 'unvoided' items */
public sumPrice(String category) {
// Get 'item' records from database with query
List<Item> items = getItems(category);
int price = 0;
for (Item i : items) {
// Bug!!! We need filtered out 'voided' items, for example, adding IF statement 'if (i.isVoided())'
price += i.getPrice();
}
return price;
}

/* Query executed by 'getItems(category)' */
SELECT
id, category, voided, price
FROM item
WHERE category = 'food';

Fig. 1. Code and query of the motivating example
requirement documents, database schema definitions, and the tendency of the records in the actual database. PPCT uses PPC (plain pairwise coverage), while SPCT uses SPC (selected pairwise coverage) for a query. PPC and SPC generate the coverage criteria using a pairwise testing technique which uses columns from the query as parameters and equivalence classes of columns as candidate values. The query results are used to verify which coverage criteria are satisfied. Both PPCT and SPCT involve six steps. In our coverage measurement tool, steps 3 - 5 are automatically implemented.

1) Execute a test case:
Testers retrieve the desired SQL SELECT query to measure the coverage from DBMS query logs or application logs.

2) Set the candidate equivalence classes of the columns in the database:
Testers can set the following equivalent classes based on the type of the column:
- **Bool**
  - Testers can set ‘true’ and ‘false’ as candidate values.
- **String**
  - Testers can set what to measure in a regular expression manner.
- **Boundary Value**
  - Testers can set the range of the values they want to measure with comparison operators (i.e., =, ≠, >, ≥, <, or ≤)

3) Execute a query:
Our tool executes the SQL SELECT query obtained by step 1 to the database in the initial input and collects the query results.

4) Generate the coverage criteria:
Pairwise testing is used as an element technology to generate the coverage criteria. Our tool generates the coverage criteria using the equivalence classes of the columns created in step 2 and the SELECT query in step 3. Details of this step are described in the next subsection.

5) Measure the coverage:
Our tool measures the coverage using the query results in step 3 and the coverage criteria in step 4. Details of this step are described in the next subsection.

6) Update the database:
According to the coverage measurement results in step 5, records are inserted, updated, and deleted in the database. Steps 3 - 6 are repeated to prepare an effective database set for testing.

B. Our Record Coverages

In this subsection, we propose our record coverages to test the database applications used in steps 4 and 5 in the PPCT/SPCT overview. Figure 3 an example of the flow for steps 4 and 5 using the motivating example in Section II.

1) Plain Pairwise Coverage (PPC):
In PPC, we focus on the all COLUMNS with TABLEs in the query.

$q$ represents a SQL SELECT query obtained by step 1 in the PPCT/SPCT overview. If $c_i(i = 1, 2, \ldots)$ represents a column in the user definition, then $C$ (the set of parameter columns) is defined as $C = \{c_1, c_2, \cdots\}$. For the example in Figure 3, the schema of the table item is shown in Table I. Because PPC focus on all COLUMNSs in TABLEs referred in the query $q$, $C = \{id, category, voided, price, created_at, updated_at\}$.

Let us refer to the equivalence classes of columns as the user definition obtained by step 2. If $d_i(i = 1, 2, \cdots)$ represents a column in the user definition, then $D$ (the set of columns in the user definition) is defined as $D = \{d_1, d_2, \cdots\}$. In Figure 3, $D = \{category, voided, created_at\}$. In the user definition, if $c_{d, k}(k = 1, 2, \cdots)$ represents an equivalence class of column $d_j$, then $E_{d_j}$ (the set of equivalence classes of column $d_j$) is defined as $E_{d_j} = \{e_{d_j1}, e_{d_j2}, \cdots\}$. In Figure 3, $E_{voided} = \{true, false\}$.

For each column $c_i \in C$, the set of possible values of $c_i$ is defined as a set of equivalence classes $E_{c_i}$, which is given by the following formula:

$$
E_{c_i} = \begin{cases} 
E_{d_j} & (c_i \in D \land c_i = d_j) \\
U_{c_i} & (otherwise) 
\end{cases}
$$ (1)

$U_{c_i}$ in formula (1) means an equivalence class that includes any possible value for column $c_i$. In Figure 3, the column id is not in $D$. Therefore, $E_{id} = U_{id}$. Because the column id is an integer type, $U_{id} = \{x | x \in \mathbb{Z}\}$. Between columns $c_i$ and $c_j \in C$, the set of combinations of possible values to be...
tested $P_{ij}$ is defined by the following formula (where $i > j$ and $N = |C|$)

$$P_{ij} = U_{c_1} \times \cdots \times U_{c_{i-1}} \times E_{c_i} \times U_{c_{i+1}} \times \cdots \times U_{c_N}$$

Then the set of all possible combinations with at least two parameters to be tested $P$ is defined by the following formula (where $N = |C|$)

$$P = (\cup_{i=1}^{N-1} \cup_{j=i+1}^{N} P_{ij}) \cup (\cup_{k=1}^{N} E_{c_k})$$

$P$ for the flow example is shown as the coverage criteria in Figure 3.

If the set of the parameter pairs appearing in the results obtained in step 3 is referred to as $\text{Appeared}(P)$, then the coverage $\text{Cov}_{\text{PPC}}$ is defined as

$$\text{Cov}_{\text{PPC}} = \frac{|\text{Appeared}(P)|}{|P|} \times 100(\%)$$

In the coverage measurement results in Figure 3, a check (cross) indicates that the pair does (does not) appear in the records. In this example, step 3 gives two records, but only three combinations, $\{(\text{false, 'food'}, \text{false, 'food'}, \text{false, 'clothes'})\}$ appear. Hence, $\text{Cov}_{\text{PPC}} = 3/18 = 16.67(\%)$. However, in this example, the parameter combination $\text{voided, category = 'food'}$ cannot appear in the records because it is filtered out by the WHERE condition $\text{category = 'food'}$. Furthermore, parameter combinations concerned with the column $\text{created}_\text{at}$ will not appear in the records because the column is not selected by the query. Therefore, PPC cannot reach 100 % and PPC is not effective in some cases.

2) Selected Pairwise Coverage (SPC): PPC focuses not only on the parameter combinations obtained by the query, but also the parameter combinations not obtained. In contrast, selected pairwise coverage (SPC) focuses on parameter combinations that can be obtained by the query to cover them efficiently. SPC regards the set of all $\text{COLUMNS}$ selected in the query $q$ as $C$. In Figure 3, $C = \{\text{id, category, voided, price}\}$. The definitions of $E_{c_i}$, $P_{ij}$, and $P$ are the same as those of PPC.

To ignore combination combinations not obtained by $q$, $P$ must be filtered. Let $W$ be the set of equivalence classes of columns satisfying the conditions in $q$. In Figure 3, $W = \{E_{\text{category}} = \{\text{'food'}\}\}$. $P_{\text{satisfied}}$, which is the parameter combinations that can be obtained by $q$, is defined as the set calculated by the algorithm in Figure 4. The dotted lines in Figure 3 show the set of combinations for $P_{\text{satisfied}}$. The set of pairs appearing in the results obtained by step 3 is referred to as $\text{Appeared}(P_{\text{satisfied}})$. Finally, the coverage $\text{Cov}_{\text{SPC}}$ is defined as

$$\text{Cov}_{\text{SPC}} = \frac{|\text{Appeared}(P_{\text{satisfied}})|}{|P_{\text{satisfied}}|} \times 100(\%)$$

In Figure 3, $\text{Cov}_{\text{SPC}} = 3/5 = 60(\%)$.

![Diagram of SPC algorithm](image)
C. Testing with our record coverage

Next, we tested the motivating example mentioned in Section II with our record coverage.

Figure 3 shows the parameter combinations in PPC. Among them, a bug is found when calling \( \text{sumPrice("food")} \) with the record where \((\text{voided, category}) = (\text{true, food})\). The necessary parameter combinations in SPC are also shown in Figure 3. Similar to PPC, a bug is detected when calling \( \text{sumPrice("food")} \) with the record where \((\text{voided, category}) = (\text{true, food})\).

IV. CASE STUDY

We carried out a case study, targeting OpenMRS [11] and Broadleaf Commerce [12]. These targets were chosen because:

1) They use SQL queries to manage the data in their databases.
2) They have real bugs, which are individually managed using a tracking system.
3) They have a sufficient number of tables and columns to evaluate RQ2.

A. Case Study Overview

OpenMRS is an open source medical record system platform for developing countries. We used the openmrs-core module, which is a core module of OpenMRS that has api and web application codes as the targets. Broadleaf Commerce is an open source e-commerce platform. We used the Broadleaf-Commerce module, which is a core module of Broadleaf Commerce. Both are written in Java™ and interact with approximately 100 tables in the database. We used MySQL® (Ver 14.14 Distrib 5.5.27, for Win32) as the RDBMS. The case study used four versions of OpenMRS and one version of Broadleaf Commerce for the control system Git [15]. Figure 5 overviews of the case study, which can be divided into five steps:

1) Systematically filter bugs:
Issues are filtered by condition. OpenMRS manages the bugs and features to be implemented in the future by JIRA [16], which is an issue management system. Initially, issues are filtered by the condition “Project=TRUNK AND IssueType=Bug AND Status=Closed”. The condition “Project=TRUNK” removes bugs not related with openmrs-core. If an issue is closed, the cause of the bug can be determined by inspecting the discussion and activities of the issue. Next issues are extracted by the committed files with at least one file related to database access. Service classes to execute SQL to obtain records, controller classes to process the records obtained through service classes, and their test classes are assumed to be related to database access. Because Broadleaf Commerce manages the bugs and features to be implemented in the future by Git issues page, issues are also filtered by the condition “label:type-bug is=closed”. Next issues are extracted by committed files as well as OpenMRS. All of these files can interact with the database directly or indirectly.

2) Manually filter bugs:
Next issues not related with bugs due to the records in the database and not detected by code coverage are manually filtered. Unrelated issues include problems in setting up software, problems generated by Continuous Integration (CI) tools, etc.

3) Reproduce bugs scenarios:
Each bug is reproduced by conducting its scenarios in order to obtain its query log. Scenarios are determined by inspecting the discussion, activities, and committed files in their issue pages.

4) Analyze bug detectability:
Each bug is then detected by three different methods: Full Predicate Coverage Testing (FPCT) based on full predicate coverage [8] (Section II), PPC Testing (PPCT) based on PPC (Section III), and SPC Testing (SPCT) based on SPC (Section III). To evaluate whether the testing methods can detect the bug, the procedure to execute each method involves three steps. First, which records cause the bug (some columns have certain values, etc.) are determined by inspecting the discussion, activities, and committed files in the issue. Second, records are prepared based on the query log obtained by the reproduction of bug scenarios so that each method achieved 100% coverage. Finally, the applicability of testing method is verified. If a testing method identifies at least one record from the first step, it is deemed as appropriate. The coverage measurement tool depends on the testing method. For FPCT, SQLFpc, a full predicate coverage rules generation tool to test SQL database queries [17], is used. Our tools mentioned in Section III are used for PPCT and SPCT. The equivalence...
reach 100% coverage vary by method. For FPCT, SQLFpc ones appear, as the cost of PPCT. The generation tools to where all parameter combinations including the redundant combinations. Therefore, we regard the number of records, the cost of the testing method. As mentioned in Section number of records to reach 100% coverage is regarded as equivalance class if the column allowes NULL values. or a not empty string. Notice that ‘NULL’ is added as an type column (e.g., char, string) is set as either an empty string (e.g., date, datetime), ‘past’ and ‘future’, which mean past or set based on the type of column. For a number-type column (e.g., integer, double, float), ‘any values in the range for the equivalence classes of columns are needed from step 4 (Section III). For user definitions, the equivalence classes for each column are.

5) Calculate costs:
Finally, the cost of each testing method is calculated. The number of records to reach 100% coverage is regarded as the cost of the testing method. As mentioned in Section III, PPC cannot reach 100% because of redundant parameter combinations. Therefore, we regard the number of records, where all parameter combinations including the redundant ones appear, as the cost of PPCT. The generation tools to reach 100% coverage vary by method. For FPCT, SQLFpc is use, whereas PICT [14], which is the test generation tool by pairwise testing with the user definitions, is used for PPCT and SPCT.

Table II shows the LOC, number of tables and columns that each version of software interacts with, and the release date of each version. Table III shows the number of bugs at each step.

### B. Case Study Results

Table IV shows the results of the bug detectability for each testing method. The first three columns under Detectability for Each Method denote the bug patterns. A check (cross) indicates that the test method can (cannot) detect the bug. The fourth through eighth columns denote the number of bugs for each version of OpenMRS and Broadleaf Commerce by pattern. The last column contains the total number of bugs for each pattern. Table V shows the cost of each testing method. The first and second columns indicate the version of OpenMRS and Broadleaf Commerce. The third through fifth columns, classes of columns are needed from step 4 (Section III). For user definitions, the equivalence classes for each column are set based on the type of column. For a number-type column (e.g., integer, double, float), ‘any values in the range for the column’ is set as an equivalance class. For a date-type column (e.g., date, datatime), ‘past’ and ‘future’, which mean past or future from the present time, respectively, are set. A string-type column (e.g., char, string) is set as either an empty string or a not empty string. Notice that ‘NULL’ is added as an equivalance class if the column allowes NULL values.

5) Calculate costs:
Finally, the cost of each testing method is calculated. The number of records to reach 100% coverage is regarded as the cost of the testing method. As mentioned in Section III, PPC cannot reach 100% because of redundant parameter combinations. Therefore, we regard the number of records, where all parameter combinations including the redundant ones appear, as the cost of PPCT. The generation tools to reach 100% coverage vary by method. For FPCT, SQLFpc is use, whereas PICT [14], which is the test generation tool by pairwise testing with the user definitions, is used for PPCT and SPCT.

Table II shows the LOC, number of tables and columns that each version of software interacts with, and the release date of each version. Table III shows the number of bugs at each step.

### B. Case Study Results

Table IV shows the results of the bug detectability for each testing method. The first three columns under Detectability for Each Method denote the bug patterns. A check (cross) indicates that the test method can (cannot) detect the bug. The fourth through eighth columns denote the number of bugs for each version of OpenMRS and Broadleaf Commerce by pattern. The last column contains the total number of bugs for each pattern. Table V shows the cost of each testing method. The first and second columns indicate the version of OpenMRS and Broadleaf Commerce. The third through fifth columns, classes of columns are needed from step 4 (Section III). For user definitions, the equivalence classes for each column are set based on the type of column. For a number-type column (e.g., integer, double, float), ‘any values in the range for the column’ is set as an equivalance class. For a date-type column (e.g., date, datatime), ‘past’ and ‘future’, which mean past or future from the present time, respectively, are set. A string-type column (e.g., char, string) is set as either an empty string or a not empty string. Notice that ‘NULL’ is added as an equivalance class if the column allowes NULL values.

5) Calculate costs:
Finally, the cost of each testing method is calculated. The number of records to reach 100% coverage is regarded as the cost of the testing method. As mentioned in Section III, PPC cannot reach 100% because of redundant parameter combinations. Therefore, we regard the number of records, where all parameter combinations including the redundant ones appear, as the cost of PPCT. The generation tools to reach 100% coverage vary by method. For FPCT, SQLFpc is use, whereas PICT [14], which is the test generation tool by pairwise testing with the user definitions, is used for PPCT and SPCT.

Table II shows the LOC, number of tables and columns that each version of software interacts with, and the release date of each version. Table III shows the number of bugs at each step.
orders.drug.concept_id. In Figure 6, a bug is detected when `order.concept_id` obtained by the left query is not the same as `drug.concept_id` obtained by the right query. However, such bugs are not detected because these values are obtained separately. A similar result occurs for the issue ‘1298’ of Broadleaf Commerce.

2) Only detected by PPCT:
The bugs in this pattern are detected only by PPCT due mainly to two reasons. First, some of the necessary predicates are omitted in the query. Second, the cause is something other than the value of the record (e.g., the number of records). Figure 7 shows an actual query with the issue ‘TRUNK-3339’. Although this query is supposed to return the number of provider records obtained by the condition, it returns the number of not-retired provider records. The module causes a bug when it is executed with retired provider records. Therefore, an additional condition ‘this.retired = 0’ is necessary. FPCT cannot detect such a bug. Furthermore, this query only selects the column `provider_id`. SPCT cannot detect this bug because the query does not select the column `retired`, which causes the bug.

3) Only detected by FPCT:
The bugs in this pattern are detected only by FPCT mainly because there is a misplaced logical operator (AND, OR) or a mistake in the conditional inversion in the query. Figure 8 shows an actual query with the issue ‘TRUNK-4530’. This query is supposed to return drug records when both the name and name of the table `concept_name` related with drug are forward matches of ‘asa’. However, this query actually returns drug records when either name was a forward match of ‘asa’. The module causes a bug when it is executed with drug records and either name is not a forward match of ‘asa’. Therefore, the logical operator ‘AND’ of the second condition in the WHERE clause should be ‘OR’. FPCT can detect such a bug because it can detect misplaced logical operators and conditional inversion mistakes in the query, whereas PPCT and SPCT cannot.

4) Only detected by PPCT and SPCT:
The bugs in this pattern are divided into two types. The first is ‘missing predicates’, which is due to two reasons; the columns whose values of records cause the bug are selected in the query or some of the necessary predicates are omitted in the query. The left side query in Figure 9 shows an actual query with the issue ‘TRUNK-4116’. Although this query is supposed to return drug records whose name LIKE %NYQUIL%, but it returns not-retired drug records whose name LIKE %NYQUIL%. The module causes a bug when it is executed with retired drug records whose name LIKE %NYQUIL%. Therefore, the additional condition ‘this.retired = 0’ is necessary. FPCT cannot detect such a bug. Both PPCT and SPCT can because the former selects the table `drug`, while the latter selects the column `retired` of the table `drug`. We obtained a similar result for the issue ‘1334’ of Broadleaf Commerce. The second type is ‘misused column values’. PPCT and SPCT detect this type of bug due mainly to two reasons; either the columns whose values of records cause the bug are selected in the query or the value of the wrong column is used in the code. The right side query in Figure 9 shows an actual query with the issue ‘TRUNK-3620’. This query selects `concept_id` and `answer_concept` from `concept_answer`. Although the value of `concept_answer` should be used in the code, the value `concept_id` is used instead. The module causes a bug when it is executed with records whose `concept_id` and `answer_concept` have different values (e.g., non-null values for `concept_id` and null values for `answer_concept`).
---

**SELECT**

... this.retired,
... this.retired,
FROM drug this
WHERE
// /g18115this.retired = 0/g18115 was redundant !!
this.name LIKE %2%
ORDER BY ... ;

---

**// At least one record obtained by the query can cause the bug !!**

**SELECT**

... FROM location_tag_map map0
INNER JOIN location_tag locationta1
ON map0.location_tag_id =
locationta1.location_tag_id
WHERE
map0.location_id = 1;

---

**5) Only detected by FPCT and PPCT:**

The bugs in this pattern are detected by FPCT and PPCT mainly because redundant conditions exist in the WHERE clause. Figure 10 shows an actual query with the issue ‘TRUNK-2392’. Although this query should return not-retired drug records whose name LIKE %2%, it returns all drug records whose name LIKE %2%. The module causes a bug when it is executed with retired drug records whose name LIKE %2%. FPCT can detect such a bug because it can notice misplaced logical operators. Similarly, PPCT also can detect such a bug because the table drug selected in the query has the column retired, which is referred to in the WHERE clause.

**6) Detected by any of the three testing methods:**

The bugs in this pattern are detected by any of the testing methods mainly because any record in the query causes the bug. Figure 11 shows the actual query of the issue ‘TRUNK-4036’. This query should return location_tag records related with the location record whose location_id = 1. However, the module causes a bug when this query returns at least one location_tag record. Because the records are prepared using the query, such a bug can be detected with any of the testing methods. A similar result is obtained for the issue ‘1310’ of Broadleaf Commerce.

---

**B. RQ2: How many records are needed to reach 100% SPC coverage?**

To reach 100% coverage for OpenMRS, 619 records for FPCT, 1643 records for PPCT, and 1437 records for SPC are needed (Table V, row 8). Thus, SPC requires about 2.3 more records than FPCT, indicating that SPC is more costly. However, compared to PPCT, SPC requires fewer records (0.87 times). For bugs that both PPCT and SPC detect, SPC is more efficient. The results indicate that although PPCT and SPCT are valuable because they detect bugs not detected by other methods, their efficiencies could be improved.

---

**C. Limitation**

**1) Cost Calculation:**

We defined the number of records to reach 100% coverage as the cost. However, for PPCT and SPCT, we need make additional efforts to set the candidate equivalence classes of the columns. In our case study, we made very few efforts because we set them uniformly based on the type of columns. In the future, we have to conduct some experiments to measure the additional cost because the equivalence classes of the columns setting takes some time in practical use.

**2) Our Coverage Measurement Tool:**

We implemented a coverage measurement tool for PPC and SPC, but some parts are incomplete.

To generate the coverage criteria in the step 4 in Section III, our tool analyzes the SQL(SELECT) query to collect the selected tables, columns, and conditions in the WHERE clause, but ignores conditions in the JOIN clause. To reduce the cost, the conditions in the JOIN clause must be collected to remove parameter combinations that do not satisfy the conditions in the JOIN clause. Actually, some queries found in the issues of Broadleaf Commerce had many JOIN clauses, but because SPCT only used the simple WHERE clause, it required as many records as PPCT. Furthermore, our tool ignores the condition between columns such as column1 = column2 in the WHERE clause. When such conditions exists in the query, focusing on the overlapping equivalence classes should reduce the costs.

In the coverage measurement in the step 5 in Section III, our tool inspects which parameter combinations are covered using the query results. However, it regards binary types and enumerated types as string types. In order to calculate PPC and SPC more accurately, both binary and enumerated types should be processed as their own types.

---

**D. Threats to Validity**

**In the case study, we manually set the equivalence classes of the columns based on the types of columns. However, this may affect the accuracy and costs of our testing methods, and is a threat to internal validity. In the future, especially in regression testing, we would like to set the equivalence classes of the columns. In our case study, we made very few efforts because we set them uniformly based on the type of columns. In the future, we have to conduct some experiments to measure the additional cost because the equivalence classes of the columns setting takes some time in practical use.**

**The targets in the case study were OpenMRS and Broadleaf Commerce, which have about 400KLOC and interact with about 1000 columns. Although its size is sufficient, Broadleaf Commerce has fewer bugs than OpenMRS. Therefore, the credibility of the results of Broadleaf Commerce is slightly lower than that of OpenMRS, and is a threat to internal validity.**

In the future, we would like to conduct a case study on another version of Broadleaf Commerce. A threat to external validity is that our case study is limited to open source software systems in a single software domain. In the future, we would like to evaluate our testing methods by applying them to industrial software with different software domains.
VI. RELATED WORK

Pan et al. proposed a method to generate a database state via dynamic symbolic execution (DSE) [18][19]. Then they leveraged DSE to examine close relationships among host variables, embedded SQL query statements, and branch conditions in the source code. They implemented their approach in Pex, which is a state-of-the-art DSE-based test generation tool. Unlike their study, which focused on increasing code coverage, our work focuses on detecting bugs, that are not detected by testing with the code coverage mentioned in Section II.

Some studies have examined coverage based on an SQL query. Chays et al. proposed a framework to test database applications [20][21][22][23] using database schema integrity constraints, user inputs, and conditions in queries to consider the state of a database. They also implemented a tool to populate a database with meaningful data that satisfy the database constraints. Tuya et al. proposed full predicate coverage to test SQL queries based on modified condition/decision coverage [7][8]. They focused on the conditions in queries to consider the necessary database state; this coverage detected misplaced logical operators or mistakes of conditional inversion. They also proposed a database shrinking technique by using this coverage [9]. Although both works focus on the conditions in the queries, they did not emphasize what is selected in the queries. However, their works provide insight some when considering the variations in the records’ values. To detect more bugs in database applications, we would like to combine the above techniques with ours.

Tuya et al. have proposed a tool to generate mutants of SQL database queries to test database applications [24][25]. They proposed four mutation operators for SQL queries: SC, OR, NL, IR. SC performs mutations on the main clauses, OR replaces the operators in the conditions, NL handles null values, and IR replaces identifiers in the query such as columns and constants. Pan et al. also applied these mutation operators to transform SQL queries into normal program codes, and subsequently generated both effective program inputs and sufficient database states via DSE to kill mutants [26]. Zhou et al. also proposed a mutation testing approach to extend these mutation operators and implemented JDAMA (Java®Database Application Mutation Analyzer) for Java®programs that interact with a database [27]. All of these work focused on SQL queries to test database applications but not the combinations of values for columns selected in SQL queries. Because PPCT and SPCT focus on the variation of column’s values selected in SQL queries, they can detect bugs even if some of the necessary conditions are omitted.

Kapfhammer et al. proposed def-use coverage for program interactions with databases [28]. They mapped create, read, update and delete, which are the four basic functions of database management systems, into database def-use. They also implemented a coverage-monitoring tool to measure this def-use coverage for various granularities [29]. However, this coverage measures whether defined objects (databases, tables, columns and records) are used but not the variations in the record’s values. In contrast, PPCT and SPCT measure the variety in the record’s values.

VII. CONCLUSION

To detect more bugs in database applications, we proposed PPCT and SPCT, which are pairwise coverage-based testing techniques. Our methods focus on the selected elements in SQL queries and apply pairwise testing techniques with equivalence classes of columns from user inputs and conditions in the queries. We also implemented a tool that measures PPC and SPC as well as automates some of the testing process. Then we demonstrated their applicability by applying PPCT and SPCT to two open source software systems, which have more than 400KLOC and interact with about 1000 columns. Although PPCT and SPCT currently have higher costs than existing testing techniques, the case study suggests that their costs can be further reduced.

To measure the costs of candidate equivalence classes of columns setting for PPCT and SPCT, we plan to conduct some subject experiments. Furthermore, we plan to consider database schema integrity constraints and clustering techniques for the values in columns in databases studied by Hashimoto et al. [30] to create initial equivalence classes of columns automatically for the setting cost reduction. Additionally, we would like to apply our techniques to industrial products.

REFERENCES


